

Bolt Beranek and Newman Inc.

LEVEL

11
AOV

12

BBN

Report No. 4143

A 072423

Development of a Voice Funnel System

Quarterly Technical Report No. 2
1 November 1978 to 31 January 1979

June 1979

Prepared for:
Defense Advanced Research Projects Agency

DDC FILE COPY

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DDC
RECEIVED
AUG 8 1979
REF ID: A641170
D

79 05 29 015

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DEVELOPMENT OF A VOICE FUNNEL SYSTEM QUARTERLY TECHNICAL REPORT NO. 2		5. TYPE OF REPORT & PERIOD COVERED Quarterly Technical report no. 2, 1 Nov 1978 - 31 Jan 1979
6. AUTHOR(s) M. Hoffman R. T. Rettberg	7. PERFORMING ORG. REPORT NUMBER 141 BN 4143	8. CONTRACTOR OR GRANT NUMBER(s) MDA903-78-C-0356, ARPA Order-3653
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street, Cambridge, MA 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 3653	11. REPORT DATE 14 June 1979
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd., Arlington, VA 22209	12. NUMBER OF PAGES 34	13. SECURITY CLASS. (of this report) Unclassified
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) 12 38P	15. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) DISTRIBUTION STATEMENT A Approved for public release; Distribution Unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Voice Funnel, Digitized Speech, Packet Switching, Butterfly switch, Multiprocessor		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This quarterly Technical report covers work performed during the period noted on the development of a high-speed interface, called a Voice Funnel, between digitized speech streams and a packet-switching communications network.		

060100

79 06 29 035

Report No. 4143

Bolt Beranek and Newman Inc.

DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 2
1 November 1978 to 31 January 1979

14 June 1979

This research was sponsored by the Defense Advanced Research Projects Agency under ARPA Order No.: 3653 Contract No.: MDA903-78-C-0356 Monitored by DARPA/IPTO Effective date of contract: 1 September 1978 Contract Expiration date: 30 November 1980 Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Deputy Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

Accession For	
THIS CR&I	X
ONE TAB	
Monitored	
Classification	
Per Mr. on file	
Distribution	
Security Codes	
Mail and/or special	
A	

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either express or implied, of the Defense Advanced Research Projects Agency or the United States Government.

DISTRIBUTION STATEMENT A	
Approved for public release;	
Distribution Unlimited	

QUARTERLY TECHNICAL REPORT II

Contents

1. Introduction	1
2. Operating System Development	2
2.1 Operating System Overview	2
2.2 Exploiting Parallelism	7
3. Switch Development	16
3.1 Parallel Data Paths	16
3.2 Switch Node Base	17
3.3 Partial Switches	22
3.4 Long/Short Messages	24
3.5 Speed Issues	25
3.6 Deadlocks	27
3.7 Error Control	30
3.8 Flow Control	31
3.9 Current Switch Design	31
4. References	33

1. Introduction

This Quarterly Technical Report, Number 2, describes aspects of our work performed under Contract No. MDA903-78-C-0356 during the period from 1 November 1978 to 31 January 1979. This is the second in a series of Quarterly Technical Reports on the design of a packet speech concentrator, the Voice Funnel.

Most of our effort during this quarter has concentrated on the development of a firm framework for supporting the Voice Funnel software, including both the elaboration of the Butterfly Multiprocessor hardware and the design of a suitable operating system for the machine, which will ease software development in a multiprocessor environment. In the following sections we present some of the results of this work.

2. Operating System Development

During the period covered by this report we have spent considerable effort developing the basic design of the operating system for the Voice Funnel. The operating system provides a user environment which attempts to insulate the application from the raw hardware, while retaining access to the capabilities provided by the hardware and not imposing much overhead. The operating system attempts to support only the application; it is not a general purpose facility and does not attempt to support program development. We will describe some of what we have learned about the appropriate design for the operating system of the Voice Funnel on the Butterfly Multiprocessor.

2.1 Operating System Overview

Experience in a variety of situations has convinced us that we should separate resource management and problem-specific concerns in the Voice Funnel software. Having built systems with and without such a separation, we are painfully aware that when no such separation exists, several significant and costly problems arise. Among the most important of these are that:

- problem-related algorithms become cluttered with details irrelevant to the problem at hand;
- development and maintenance proceeds much more slowly than the intrinsic difficulty of a task would suggest;
- adaptation to changes in strategy is hindered by the inflexibility of the general structure; and

- errors frequently occur which are not directly related to the problem being solved.

With such strong arguments in favor of this separation, it may seem surprising that the alternative is ever considered. There are, of course, dangers, both apparent and real, in creating such a separation. The separation is typically achieved by building an operating system. An operating system is seductive and, if not controlled, it can grow to consume far too much of the personnel, memory, and processor bandwidth available to a project. Also, because it is a separately identified component, an operating system can appear to represent additional or unnecessary work. Of the two concerns, the latter is more apparent than real, since what really occurs is that work which would otherwise be distributed throughout the software is instead consolidated and shifted from one area into another. The concern about the operating system task growing out of control is real, however, and must be carefully managed.

The approach we propose to take is to build a limited, but potent and extensible, operating system which is sufficient to meet the perceived requirements of the Voice Funnel and which is flexible enough to adapt to changes in those requirements and to the emergence of any future requirements which can reasonably be anticipated. We have begun this task by reviewing those capabilities which have become common in operating systems

[BRIN 73] and by examining any special requirements which are imposed by the Voice Funnel task or the Butterfly Multiprocessor hardware. We have attempted to select those capabilities needed in one form or another for the Voice Funnel and to design an operating system which provides those capabilities while showing promise of being extended to meet additional requirements, should they arise.

Throughout the design of both the Voice Funnel and the operating system software, we have paid considerable attention to the choice of services to be provided by the operating system, for it is not the number or complexity of the operating system features which matters, but rather their simplicity, power, and cost. Time spent in reducing the overall conception to a small set of elementary, powerful, and inexpensive facilities will pay off handsomely in reduced time to develop both the application and the operating system, in reduced consumption of processor cycles, and in increased effective utilization of personnel and hardware.

Three concepts in particular have guided us in pursuing these objectives. We have tried to be conscious of what the programmer can do himself as easily as can be done automatically. We have tried to avoid building attractive facilities for which we cannot see a clear justification in the current application. Finally, we have taken advantage of the fact that the Voice

Funnel is a dedicated application. This will allow us to avoid much of the checking and enforcement which would be required in an environment which had to run arbitrary user programs. Instead, we will be able to limit checking to those conditions which might be expected to arise from routine hardware or software failures rather than from malicious behavior or outright negligence.

The principal facilities of the proposed operating system are briefly described in the following paragraphs.

- Processor Management (or Scheduling): Allows a large number of processes* to share one or more processors without requiring any of them to know about multiple processors, about which processor they might run on, or about what to do with time they are not using. Assigns work to processors, shares available processor time based on a mix of priority and fairness considerations, and attempts to maintain more or less level load on the various processors. Permits processes to run concurrently (i.e., overlap in time), and permits them to operate asynchronously (by buffering or queuing work or data passing between them).
- Memory Management: Ensures that each process has transparent access to the memory assigned to it, allows protection of the private memory (e.g., instructions and stack) of any process from any other process, and allows voluntary sharing of memory by cooperating processes. Relieves each process of the task of explicitly managing the memory mapping registers, and protects each from incorrect use of the registers by the others.

* We will use process in its customary computer science sense to refer to the various independent (and perhaps cooperating) software components (or programs) which may be given control of a processor by a scheduler. We will use task (which some computer scientists have used in place of process) in its everyday sense to mean a piece of work to be done.

- Inter-process Communication: Facilitates the transfer of messages or data from one process to another, either through shared memory or by copying data from one address space to the other. Provides simple send and receive mechanisms which relieve the user of the need to perform mutual exclusion, memory management, and scheduling operations directly.
- Process Synchronization: Provides for coordinated use of shared data, preventing conflicting access or modification by multiple processes. Provides for coordination between processes awaiting or causing common events.
- Interrupt Handling: Provides a common mechanism for servicing interrupts and for utilizing them to keep work flowing briskly through the system.
- Reliability and Availability: Provides mechanisms for (1) periodically assessing the health of all hardware, all software and all critical data bases; (2) identifying and removing from use any failed components; and (3) realigning the remaining components to continue operation at reduced capacity (graceful degradation). Presents a reliable framework within which the application can run without itself having to attend to all of the details.

A framework such as we have described presents a number of important advantages. The Voice Funnel software will be organized by function, with capabilities that are required at many points being concentrated into a small number of elementary and powerful functions which can be built and tested once and then used repeatedly. Problems intrinsic to the application will be clearly separated from problems of taming the environment, with the result that solutions of both types can be developed and expressed independently. This will yield simpler and more effective solutions in both areas. It will make it easier to experiment with new algorithms or to incorporate new

requirements, since the number of problems requiring simultaneous solutions will be reduced. It will also increase the safety of critical data, since the data will be handled in fewer places. While there will certainly be some cases in which this solution will be less efficient than one in which the application handles all resource problems in line, careful attention to the potential dangers of this approach will enable us to minimize them. The gains realized through more effective global control and optimization, through greater flexibility and adaptability, and through greater reliability, will yield benefits much greater than the occasional inefficiencies incurred.

2.2 Exploiting Parallelism

Numerous workers have attempted to achieve parallelism at various levels in the program decomposition hierarchy [BAER 73]. This has been attempted at the following levels, among others:

- at the program level (which has occurred most often),
- at the procedure level (for example, PL/I for the IBM 370),
- at the statement level (for example, the parbegin/parend construction of Dijkstra [DIJK 65], or the ALGOL 68 compiler for CMU's C.mmp multiprocessor [KNUE 76]),
- at the sub-expression level (primarily in theoretical work), and
- at the instruction level (for example, the CDC 6600 [THOR 64]).

Of course, at each level, there may be sequences of objects (e.g., a group of statements which must be executed sequentially) for which ordering must be preserved.

For various reasons we have chosen to focus at the program level in providing parallelism within the Voice Funnel. The application contains enough parallelism at this level to take effective advantage of our multiprocessor architecture. In addition, this approach provides good control over locality of memory references, which greatly influences the level of hardware efficiency achieved. Finally, by using this approach, we can use an existing compiler (with a new Z8000 code generator), rather than having to write a new compiler or undertake major structural changes to an existing one.

Adopting the second approach (procedure level parallelism) would require modifying the compiler to use some mechanism other than a stack for automatic variables, subroutine linkage, and temporary storage, since multiple processors could not share a stack. In addition, whenever multiple processors were applied to procedures within a program, any data inherited from the calling procedure, and perhaps the instructions, would be remote to the new processors. For the Butterfly Multiprocessor, this might have significant adverse effects on execution efficiency if the ratio of remote to local references became too large. Finally, the task of writing programs to utilize this degree of parallelism would be more difficult.

The third approach (statement level parallelism) would have all of the difficulties associated with the procedure level approach and would place additional burdens on the programmer and the compiler. At this level, there are two principal sub-approaches available: having the compiler recognize and exploit opportunities for parallelism, and having the programmer do it [DIJK 65]. The former sub-approach obviously requires both a clever compiler and a clever run-time system [KNUE 76]. Both approaches, however, require that storage management similar to that required for parallel procedures be employed wherever the compiler or the programmer introduces parallelism (i.e., much more often than would otherwise occur). Programming at this level would be much more difficult than at earlier levels because the programmer would have to write true multiprocessor programs, rather than cooperating uniprocessor programs or procedures.

Work on the fourth approach (sub-expression parallelism) has been largely theoretical [BAER 73] and is not of interest here. Obviously, it requires a sophisticated compiler, and it may well require special processor capabilities. Because the burden of detecting and exploiting parallelism would be shifted from the programmer to the compiler and/or the processor, the programming task would be easier than for the previous approach.

The fifth approach is not really a multiprocessor approach. Instead, it has been applied to large uniprocessors with multiple

function units capable of executing several instructions simultaneously. Like the fourth approach, it is not of real interest here.

A second major choice, in addition to selecting the level at which to apply parallelism, concerns control mechanisms to be employed for switching control between parallel streams. In particular, it concerns whether the system scheduler should be involved in all decisions to switch control, or whether some control switching decisions might be made more efficiently by closely cooperating streams within an application. The former approach has usually been taken, and in such cases the parallel streams have normally been called processes. More recently, certain workers [KNUE 76] have suggested another level of scheduling in which each process might be allocated one or more processors which it would then switch amongst various internal activities using much simpler mechanisms than a scheduler might use.

We consider the concept of scheduling activities to be a very intriguing area for further study. However, we will not use it for the Voice Funnel because it seems most appropriate when parallelism occurs at the statement level or the procedure level and when the frequency of non-local references does not strongly affect system efficiency.

A quite different attempt at eliminating the scheduler was used in the Pluribus Multiprocessor [KATS 78]. In the Pluribus, programs are written as strips. Strips have two very important properties: they must complete within a very limited time (determined by device latency requirements), and they may not preserve any private context when they finish (since they have no context, context switching is avoided). Work to be done by the strips consists of either I/O device service or work that has been queued internally. Each device and each queue has a priority. Extremely fast dispatching is provided by a hardware unit from which a processor which has completed a strip can determine the identity of the highest priority device or queue requiring service. Despite the advantages of rapid dispatching and minimal context switching, the Pluribus approach has proven less flexible and more difficult to program than the other approaches considered. In fact, recent work on the Pluribus has involved superimposing a process mechanism upon the strip mechanism.

Much of the foregoing has been independent of the specific characteristics of the Voice Funnel. It would, therefore, be appropriate to consider why the choices made thus far are sensible for that application.

The Voice Funnel will comprise a large number of independent data streams going to or from various speech terminals. Each

data stream will comprise a large number of items (control requests and speech packets). Each item will pass through a number of processing stages (speech terminal input, one or more multiplexing stages, output to the PSAT, and the reverse), both as an individual item and as a member of an aggregate. While end-to-end sequencing within data streams must be observed, sequencing between data streams need not be.

The work to be done by the Voice Funnel thus breaks down naturally into a number of tasks corresponding to the processing of a single item or aggregate of items at a particular stage. Except that any item must move through its stages in sequence, and except that items from the same data stream must be delivered in sequence at their destination, it will generally be possible to perform tasks in parallel. Moreover, because processing stages will typically be very small, further decomposition of tasks will not usually be attractive because even negligible overhead would become large as task size became very small.

Given this breakdown of the work, it is natural to view the application software as comprising a number of programs each of which will be able to perform one of the tasks which make up the application. Because we have initially chosen to allow parallelism only at the program level, and because we do not intend to allow processes to perform sub-scheduling (i.e., we will not utilize activities), each program servicing a task will be a process.

In order to achieve parallelism between data streams, it will be necessary to process several tasks of the same type simultaneously on a number of processors. This will require several instances of each process for which parallelism is desired, since only one processor at a time can run a single instance of a process.* While all such instances might share a copy of the code, processor utilization will be much more efficient if code is always local to the processor on which it runs. Therefore, each processor which will run a process will have a copy of the code. Since each instance of a process must have its own stack and private data, these will also be replicated, and for efficiency reasons will also be local to the processor on which they will run. By making multiple copies of all processes, whether or not their instances will actually run in parallel, we will provide the scheduler with more choices as to where to run each process, and we will provide redundancy as protection against the loss of any single processor.

Most Voice Funnel processes will be created when the application starts and will cycle indefinitely, processing tasks successively as they arrive for service. In addition, it will be

* Where it is necessary to distinguish between them, we will refer to the individual instances of a process as instances or as specific processes, and we will refer to the process in general as the generic process. Similarly, we will refer to individual occurrences of a task as occurrences or as specific tasks and to the task in general as a generic task. Recall that processes are instruction streams and that tasks are units of work.

possible to create or remove processes at any time it is desirable to do so. We have chosen to create processes in anticipation of need, rather than creating processes on demand (as would be done for a time-sharing system), because we know in advance which tasks will occur and that they will recur at frequent intervals. We also do this because the Voice Funnel cannot afford either the delay from loading code or the delay from allocating a stack and private data and updating the scheduling tables each time a task arrives for service. Because the most commonly occurring tasks will be very brief and will be performed without any intermediate waits, there will be no advantage to interleaving the execution of two instances of one process on the same processor.

Because the Voice Funnel will be a dedicated application, we have endeavored to avoid requiring that the operating system contain facilities which would add unnecessary overhead or delay to Voice Funnel operation. On the other hand, we have attempted to define operating system facilities which can be provided in a manner which will permit their use in other situations. Thus, while we anticipate that Voice Funnel processes will not have to share the system, alternative applications (such as a time-sharing system or another dedicated application) could use the same facilities (perhaps with elaboration) to accomplish different ends. Thus, if it suited a particular application,

processes might terminate voluntarily rather than cycling, multiple copies might not be loaded, and processes might be loaded only when needed.

3. Switch Development

While the basic design of the Butterfly Switch has been previously developed and described [BBN 78], many of the parameters of the Butterfly Switch have been closely examined in the past quarter. These variations on the basic Butterfly Switch should have a major impact on the performance of the Butterfly Multiprocessor.

This section presents some of the more interesting variations of the Butterfly Switch which we have explored. After this presentation, we will summarize the switch as we would implement it now, although further development may cause the actual implementation to differ from this definition. The design of a Butterfly Switch is complex enough that many of the topics interact. We therefore apologize in advance for discussing some topics before they have been properly introduced.

3.1 Parallel Data Paths

Clearly, the bandwidth of the Butterfly Switch is going to be an important parameter of a Butterfly Multiprocessor. The simplest way to improve this bandwidth is through parallelism in the data paths of the switch. This also has the advantage of amortizing the overhead of the control portion of the switch -- the control logic in an MSI implementation, and the control pins in an LSI implementation. At times, we will use the term

thickness as a synonym for data parallelism when referring to the dimensions of a Butterfly Switch.

It is not necessary to go to extremes, when introducing parallelism, by designing switches which permit the transaction to be only one tick long; a small amount of parallelism (such as 2 or 8 bits parallel) produces a switch in which one path has a bandwidth of many tens of megabits. Large amounts of parallelism become unattractive when the marginal system advantage of an extra data path becomes less than the marginal increase in the cost of the switch.

3.2 Switch Node Base

Previous descriptions of the Butterfly Switch have assumed that the switch nodes have two inputs and two outputs. This choice is somewhat arbitrary; in fact, a switch node with any number of inputs and outputs (greater than 1) could be used. Indeed, we feel that a switch node with 4 inputs and 4 outputs is much better because it reduces the number of switch nodes by a factor of 4 and the number of interconnections by a factor of 2. Indeed, a switch of base 8 would reduce the number of nodes further. However, as we will see, large bases lead to less modular switch sizes and larger switch node implementations. A choice of 4 for the base seems a good compromise.

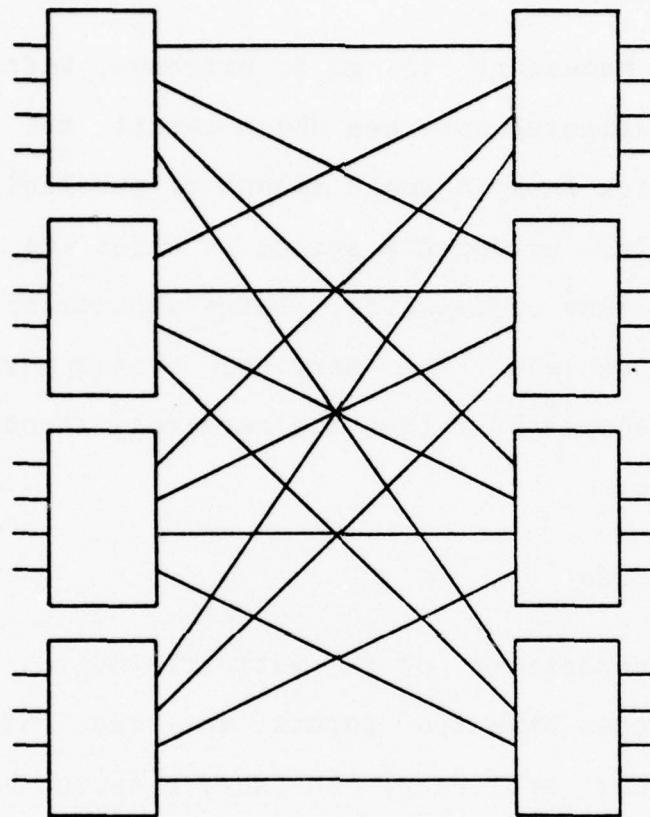


Figure 3.2-1 A Butterfly Switch of Base 4

We will call a switch node which has B inputs and B outputs a switch node of "Base- B " because the number of nodes in a Butterfly Switch with N ports is:

$$(N/B) * \log[B](N)$$

Unfortunately for bases other than two, the easily recognizable structure of the Fast Fourier Transform is lost. For example, Figure 3.2-1 shows the interconnection pattern for a 16 X 16 Butterfly Switch constructed from base-4 switch nodes. For comparison, Figure 3.2-2 shows a 16 X 16 Butterfly Switch using base-2 Switch Nodes.

From these two figures, it is obvious that the switch with the higher base has fewer nodes and fewer wires. We can quantify the impact of the selection of a base on the size and structure of the switch in hopes that this will lead us to an optimum base for the Butterfly Switches we will construct. These calculations assume that the number of ports is an integral power of the base of the switch. Other numbers of inputs and outputs will be discussed later. The calculations count the number of input port wires but not the number of output port wires. The difference is N wires.

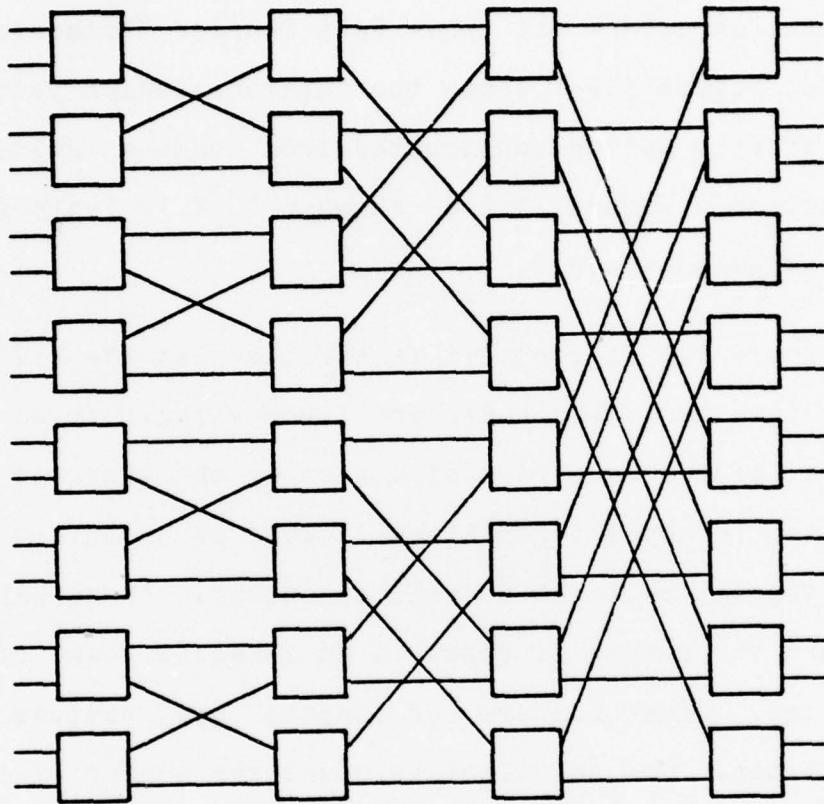


Figure 3.2-2 A Butterfly Switch of Base 2

Assume:

S = Number of switch nodes
N = Number of ports into the switch
B = Base of the switch
C = Number of columns
W = Number of interconnections

Then,

$$\begin{aligned}C &= \text{LOG[Base B]}(N) \\S &= (N/B) * \text{LOG[Base B]}(N) \\W &= B * S\end{aligned}$$

If we wish to compare switches of two bases, we can derive the ratios of the number of switch nodes and the number of wires in two switches as follows, assuming an equal number of input ports:

$$S_1/S_2 = (B_2 * \text{LOG}(B_2)) / (B_1 * \text{LOG}(B_1))$$

$$W_1/W_2 = (\text{LOG}(B_2) / \text{LOG}(B_1))$$

The following table compares base-2 switches with higher base switches. In each case, the table entry gives the improvement provided by the higher base. For example, a base-2 switch has 12 times as many switch nodes as a base-8 switch.

BASE	S1/S2	W1/W2
2	1	1
4	4	2
8	12	3
16	32	4

Thus a switch with a larger base has fewer switch nodes and fewer interconnections. Switches with large bases have a further advantage over those with small bases in that a single switch node can be built as a B X B crosspoint switch. This reduces the number of conflicts in the network, thereby increasing the performance of the switch.

While the reduction in the number of collisions is an important advantage of larger bases to the performance of the network, we should also note that a higher base implies fewer columns. This will decrease the actual delay across the switch to a small extent.

While these arguments suggest that the largest possible base should be selected, there are problems with large bases. In particular, the Butterfly Switch does not grow as smoothly as we had earlier expected. Although it is practical to make a large range of switch sizes from the same switch node, there are rather sharp growth points where the size of the switch must be increased significantly to add one more port. We will discuss these issues more in the next section.

In light of these considerations, we expect to build our switch using base-4 nodes since it achieves a good compromise between the complexity of the switch node and the improved performance of the switch.

3.3 Partial Switches

As we have noticed before, Butterfly Switches have certain preferred numbers of ports which result in complete switches. By complete, we mean that in the switch, there are no nodes which have unused inputs or outputs. These numbers are a function of the base. The number of ports in a complete switch is B^{*i} where

B is the base and i is an integer. The progression for various bases is as follows:

Base	Number of Ports (N)											
2	1	2	4	8	16	32	64	128	256	512	1024	...
4	1		4		16		64		256		1024	...
8	1			8			64			512		...

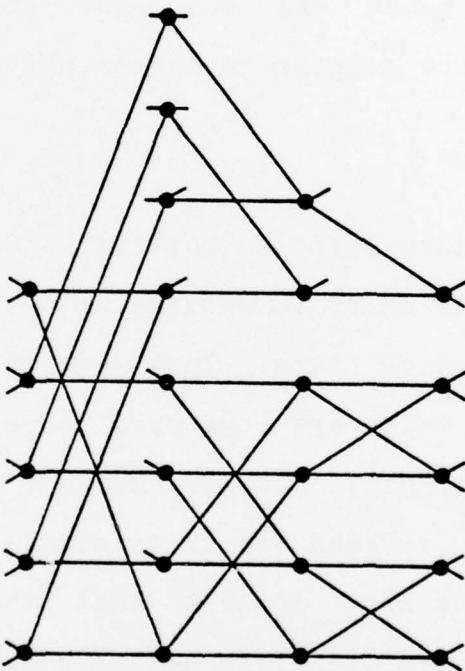


Figure 3.3-1 A Partial Switch

If N is not an integral power of the base, then the structure of the switch is that of a switch which is "the next size larger": that is, one with B^{**i} ports where i is the next larger integer. This leaves a switch which has many unused ports, and potentially even unused switch nodes. We can remove the unused switch nodes from the switch network.

Figure 3.3-1 is an example of a 10 X 10 switch which has had the unneeded switch nodes removed. The pattern of unneeded switch nodes is much more complex in larger switches.

3.4 Long/Short Messages

The pipelined structure of a Butterfly Switch implies a relatively large initial delay in setting up a transfer, followed by a very high transfer rate. This means that the Butterfly Switch favors messages which are long over those which are short. The structure of a tightly coupled multiprocessor, however, requires the ability to read and write single words across the switch. As a result, we have decided that the hardware will support both single word transfers and multiple word transfers. This decision is a matter of application of the switch, since the switch need not discriminate between these two message types.

The presence of long messages in the switch may adversely affect latency of the short messages. Indeed, a transfer of, say, 64 kilowords, would take about 26 milliseconds, during which

one path through the switch would be constantly occupied. We would expect this to block many short messages and stop many other processors.

The simplest solution to this is to divide long transfers into shorter pieces, of say 16 words. This has four further advantages:

1. The switch interface implementation is simplified because the messages can be assembled in hardware buffers. This also decouples the switch rate and the processor node rate.
2. With an appropriate switch interface design, the local processor can access remote memory locations during the transfer of a long data block.
3. Shorter messages improve throughput when switch transmission errors occur.
4. Should the processor need the full bandwidth of the memory for fast interrupt servicing, the sectioning of long data blocks into short data blocks may allow the processor to temporarily stop a data transfer between the small blocks during the interrupt service routine.

3.5 Speed Issues

For the Voice Funnel application an important speed issue is the bandwidth of a Butterfly Switch. This bandwidth is a function of five factors:

1. Switch thickness - The bandwidth is linearly related to how many bits move across the switch during one clock period.
2. Switch clock frequency - The bandwidth is linearly related to the switch clock frequency.

3. Control overhead - As each message consists of data bits and control bits, the bandwidth is related to the ratio of data bits to message bits (i.e., data bits plus control bits).
4. Memory bandwidth - At some operating point of the above four factors, the data cannot be written into or read from the local memory fast enough to keep the switch path busy and still permit the local processor an occasional access to its memory.

The switch clock frequency is limited by the time it takes to establish a connection between a switch node input and output port or by the time it takes to propagate the data to the next switch node. In very large switch configurations, the propagation time may be larger than the connection time. We have estimated the maximum clock rate using standard TTL Schottky MSI components. The worst case connection time is 77 nsec and the worst case propagation time is 40 nsec. Because of the 4 MHz maximum clock frequency of the microprocessor, a 12 MHz switch clock frequency seems to be a good choice.

Because not every switch clock period transfers data, the potential bandwidth of 48 MHz is further reduced. For the unidirectional switch, the table below shows the reduction in bandwidth due to the inclusion of control bits in each message for several data block sizes: (Base-4, 128 processors, thickness of 4).

Data Bits	Message Size	Effective Bandwidth (MHz)
-----------	--------------	---------------------------

16	96	8
32	112	13.7
64	144	21.3
256	336	36.6

As can be seen, data sent or retrieved from remote memory in multiple word blocks makes much better utilization of the Butterfly Switch.

3.6 Deadlocks

Consider a switch interface having a single input message buffer and a single output message buffer. The deadlock shown in Figure 3.6-1 can be imagined as resulting from the following scenario: 1) both Node A and Node B send memory "request" messages to each other simultaneously; 2) after these requests have been processed, but before the replies have been sent, requests from two other nodes (X and Y) arrive at the inputs. At this point, the "replies" in Node A and Node B cannot be sent because the receive buffers are occupied by the foreign requests. Yet, no further processing of requests can be done to free the receive buffers until the replies have been sent -- the system is locked up.

However, if we can assure that a reply message can always be sent and accepted, the system is free of deadlocks. This guiding principle will help us understand the requirements on the design of the switch and the switch interface which are necessary to

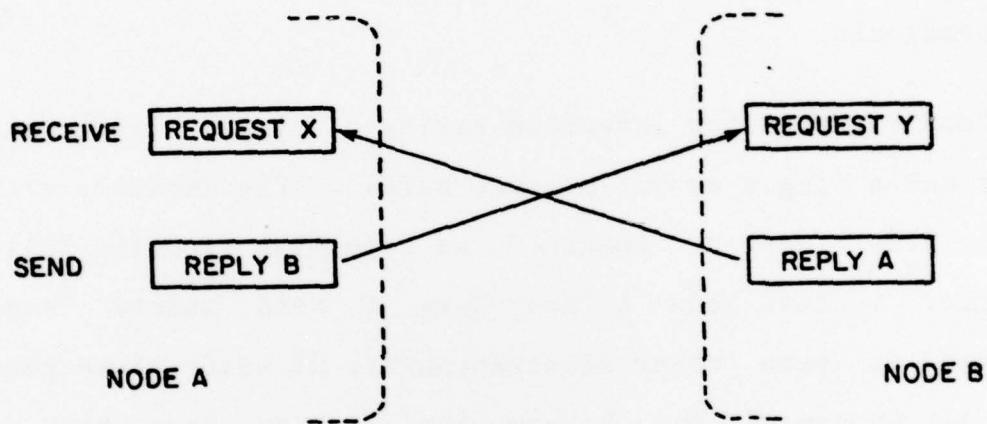


Figure 3.6-1 A Deadlock

assure deadlock-free operation. To validate these ideas, the simulations which we will be describing later have implemented these mechanisms and have not exhibited deadlocks.

If we are to assure that replies are always accepted, we must provide a mechanism which accepts them even if the receiver buffer is occupied. To assure this, we provide two buffers: one for requests and one for replies. This is not enough; we must also have a way to peek at each message and turn requests away when the request buffer is occupied so that replies can always be accepted at the entry port to the processor node. This function can be performed by first accepting the header of the message and then asserting the REJECT signal if it is a request and the request buffer is busy.

Note that it is this requirement which leads to deadlocks when the Wait strategy is applied to the one way switch. The Wait strategy has no mechanism for turning away requests and letting replies through.

If we are to assure the path of a reply, we must also assure that it can be transmitted in the first place. The straightforward way to do this is to provide two transmit buffers and allocate one for replies only. Then, when a request is rejected, before retransmitting the request, the transmitter should check the reply buffer and if a reply is present, send it instead of the request.

Finally, we are counting on randomness in the switch to assure that replies do not repeatedly encounter the same or other

request messages in the switch, thus also leading to deadlocks. This is a level of detail which we have not yet explored.

3.7 Error Control

We have used the analogy of the Butterfly Switch as a communications network before. As we know, one of the important characteristics of a communications network is its handling of errors. We expect two things of such a network: 1) extremely low probability of undetected errors, and 2) automatic recovery from errors.

Although the Butterfly Switch is within a computer system, we should still expect errors to occur -- if not because of noise, then because of either intermittent or solid component failures. We have discussed the introduction of extra columns in the switch to provide extra paths which will permit operation once failed components have been identified. In order to detect errors when they occur, we expect to provide check bits on each transaction.

Errors in the data of a message will be detected by these check bits, but errors in addressing will not be. A simple way to detect addressing errors is to include the destination address in the checksum when the message is sent and to have the destination processor node include its own address when it verifies the checksum. Thus, if the message reaches the correct

destination without error, the checksum will be correct. Otherwise, an error will be detected. This method has the advantage of avoiding the transmission of the destination address in the text of the message.

3.8 Flow Control

Another characteristic which we have come to expect of communications networks is a facility for flow control. This is necessary whenever we cannot guarantee that the receiver has sufficient resources to accept what the transmitter may wish to send. Our initial design for the switch interface implies a very structured buffering arrangement with few message format variations. As a result, our initial design would not need flow control in the switch.

3.9 Current Switch Design

In summary, we felt that the best switch design for the range of machines currently anticipated is as follows:

1. Conflict Resolution - We will use the "retreat" strategy.
2. Parallel Data Paths - Data parallelism of 4 bits (called a "nibble").
3. Switch Base - Base-4.
4. Extra Columns - None for a 16 input switch, perhaps one for a 64 or 256 input switch.

5. Long/Short Messages - Both long messages (multiple word transfers) and short messages (single word transfers) will be supported. To keep the latency of short messages low, long messages will be broken into blocks of 16 words or less.
6. Speed - We expect the switch to clock at between 10 and 12 MHz for an effective bandwidth of 40 to 48 MHz per path. In a switch with 256 ports, this implies a maximum aggregate data rate of 10 to 12 gigahertz.
7. Deadlocks - We will avoid deadlocks by using the "retreat" strategy in the switch, and by designing the switch interface so that it can always accept a reply.
8. Error Control - The switch itself will not perform any error control, but the switch interfaces will use check bits to detect errors in data or routing.
9. Flow Control - None in the switch. The sender and receiver will be designed so that the transactions do not need flow control.

4. References

- [BAER 73] J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing", Computer Surveys, Vol. 5, No. 1, March 1973.
- [BEN 78] M. F. Kraley and R. D. Rettberg, "A New Multiprocessor Architecture", Bolt Beranek and Newman Inc., Report 3501, December 1978.
- [BRIN 73] P. Brinch Hansen, Operating System Principles, Prentice-Hall, 1973.
- [DIJK 65] E. W. Dijkstra, "Cooperating Sequential Processes", Technological University Eindhoven, The Netherlands, 1965. (Reprinted in Programming Languages, F. Genuys, ed., Academic Press, 1968).
- [KATS 78] D. Katsuki, E. S. Elsam, W. F. Mann, E. S. Roberts, J. G. Robinson, F. S. Skowronski, and E. W. Wolf, "Pluribus -- An Operational Fault-Tolerant Multiprocessor", Proceedings of the IEEE, Vol. 66, No. 10, October 1978.
- [KNUE 76] P. Knueven, P. G. Hibbard, and B. W. Leverett, "A Language System for a Multiprocessor Environment", Proceedings of the Fourth International Conference on the Design and Implementation of Algorithmic Languages, Courant Institute of Mathematical Sciences, Computer Sciences Department, New York University, June 1976.
- [THOR 64] J. E. Thorton, "Parallel Operation in the Control Data 6600", AFIPS Conference Proceedings 26, 1964.

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency

Dr. Robert E. Kahn (2)

Defense Supply Service -- Washington

Jane D. Hensley (1)

Defense Documentation Center (12)

Bolt Beranek and Newman Inc.

Library

Library, Canoga Park Office

R. Brooks

P. Carvey

P. Castleman

F. Heart

M. Hoffman

D. Hunt

M. Kraley

W. Mann

R. Rettberg

E. Starr

D. Walden

E. Wolf